Foundations of Information Processing

Making algorithms: fundamental parts

Let us think
how to make better algorithms

Consideration 1:

How *creative results* of problem solving could

be utilized in

as a *wide* manner as possible?

Consideration 2:

Assuming the algorithm for a certain problem already *exists*

could *all the corresponding* problems

be solved using this algorithm?

Consideration 3:

How the algorithm should be *presented* that it is *understandable*

to its *user* and

others who *develop* algorithms?

Towards making algorithms efficiently

Well defined and formalized problems can be solved "mechanically" with algorithms, with certain assumptions.

Making algorithms requires creativity, but executing the algorithms is a systematic operation.

Making algorithms: fundamentals to be learnt

”I am rarely happier than when spending entire day programming my computer to perform automatically a task that it would otherwise take me a good ten seconds to do by hand.”

(Douglas Adams, an English writer

who wrote The Hitchhiker's Guide to the Galaxy)

- **Refining algorithms:** extending each part of the algorithm step-by-step gradually into subtasks.
- **Modularity:** an independent solution of a subtask (a module).
- **Imperative paradigm:** a paradigm of computer programming where the program describes steps that change the state of the computer, presenting the algorithm as sequences of instructions.
- **Pseudo language:** a tool to present the algorithm as a program.

Algorithms: required properties

Source: J. Boberg, Johdatus tietojenkäsittelytieteeseen, Turun yliopisto, 2010 (in Finnish).
Source:E. Horowitz et al. Fundamentals of Data Structures in C. Computer Science Press, 1993.

- **Generalizability**: suitable to all the cases of the task.
- **Determinism**: the solution must be deterministic and at each step it is known unambiguously what to do next.
- **Output**:
  - Correctness: the result is always correct.
  - Finiteness: the algorithm always terminates.
- **Input**: the range of the input and how it affects the result.
- **Effectiveness**: how robust and feasible of each step of the solution is as a function of computing time and as a function of the use of space.

Algorithms: gradual refinement and modularity

Source: J. Boberg, Johdatus tietojenkäsittelytieteeseen, Turun yliopisto, 2010 (in Finnish).

- **Reduction** ≈ dividing into parts:
  - Divide the task/problem into parts.
  - Solve these subtasks.
  - Present the solution of the whole problem by merging the solutions of the subtasks.
- **Gradual refinement**:
  - Repeat the diving into more detailed parts as needed (algorithm comprehensibility).
- **Modularity** ≈ diving into general purpose parts:
  - Present those parts of the algorithm as modules which are useful for general purpose so that it is simple to use them in solving other problems.

Our example algorithm:
Let us help Mr. Kottarainen more specifically

Let us revisit our example algorithm for Mr. Brian Kottarainen:

```
1. Gather the dirty socks.
2. Go to the washing room.
3. Wash the socks.
4. Dry the socks.
5. Pick up the socks from the washing room.
```

The algorithm can be refined gradually and using modules.

Let us see next how.

Example algorithm: refining gradually and using modules

Version 2:

```
1. Gather the dirty socks.
2. Go to the washing room.
3. Wash the socks.
   3.1 Sort the socks as dark and
       white ones.
   3.2 Place the socks into
       different washing machines.
   3.3 Start washing.
   3.4 Wait until the washing
       program ends and take the
       socks out of the machine.
4. Dry the socks.
   4.1 …
5. Pick up the socks from the
   washing room.
```

Subtasks can be presented as modules of general purpose to increase usability. Thus, no need to rewrite the code, just making a call in the algorithm is enough.

```
Module ”Start washing”
1. Add a suitable
   amount of washing
   liquid.
2. Close the lid.
3. Select the washing
   program.
4. Press the start
   button.
```

Imperative paradigm: the algorithm as sequences of instructions

Source: J. Boberg, Johdatus tietojenkäsittelytieteeseen, Turun yliopisto, 2010 (in Finnish).

- A typical way to present algorithms is to write algorithmic steps as textual *sentences*.
- In the imperative paradigm the steps are presented as imperative sentences (verbs as imperatives) = *instructions*.

```
Gather the dirty socks.
```

- Algorithms consist of *procedures* and *functions* = *modules* of sentences.

```
”Start washing”
```

Pseudo language for learning to make algorithms

Source: J. Boberg, Johdatus tietojenkäsittelytieteeseen, Turun yliopisto, 2010 (English).

- A pseudo language is a “semiformal” language to learn the typical structures of algorithms, implemented as a pseudocode program.
- The goal is to pay attention at
  - principles of making algorithms and
  - meaning of sentences (semantics),

  instead of strict grammar (syntax).
- The pseudo language is not directly applicable to programming, but the matters to be learnt can be applied
  - to understanding principles of a real programming language and
  - to programming itself in practice.
- Each programming language shares roughly the same principles.

Imperative paradigm: parts of the algorithm

Source: J. Boberg, Johdatus tietojenkäsittelytieteeseen, Turun yliopisto, 2010 (in Finnish).

- Next, the parts of the algorithm are considered.
    - The pseudo language is being used in examples.
- **Statement**:
    - A syntactic unit of an imperative programming language that expresses some action to be carried out in the algorithm (instruction).
    - This relates to the sentence in the written language.
- **Expression**:
    - A part of the statement which produces a value.
- **Variable**:
    - An allocated position in the memory to store data.

Parts of the algorithm: assignment statements

- **Assignment statement** in the pseudo language:

  `variable:=expression`

- Arithmetic operators are as follows: `+ - * /`
- For example, `x:=x+2*sum`

  where `x` and sum are variables with the value.
- The new value of `x` is `the current value + 2*sum`.
- If `x=3` and `sum=4` then `x=3+2*4=11`.
- Each variable contains the type of the variable (data type):
  - Integer: 0,1, 2, 3, …
  - Float/real: 3.562 (values other than whole numbers)
  - Character: a, b, c, …, A, B, C, …
  - Boolean: true/T or false/F.
  - String: text as characters, e.g., “word”.

Parts of the algorithm: execution order

Order of executing statements:

- **Sequential**:
  - Consecutively statement by statement.
  - This is to be learnt in this course.
- **Parallel**:
  - Several statements in parallel at the same time.
  - Parallel computing, parallel processors.
  - This is clear
    - when statements are independent to each other or/and
    - the same statement can be applied to the data in parallel.

Parts of the algorithm: execution order (cont.)

Something else, for example, **declarative**:

- For example, logical programming (Prolog) where Boolean values of logical expressions determine the execution order.
- This means that the value (true or false) tells where to go next in the program.
- Example:

```
sibling(X,Y):-parent_child(Z,X),parent_child(Z,Y)
```

  where `sibling(X,Y)` becomes true only, if `parent_child` and `parent_child` are true with the given `Z`, `X`, and `Y`.

- Then, the value "true" of `sibling(X,Y)` enables the execution of the next statement in the program where `sibling(X,Y)` is part of.
- So, this place is known only after the value (true or false) has been computed.
- Just an example: learning Prolog is not a part of this course.

Parts of the algorithm: control statements

Control statements:

- Sequential execution:
  - After the execution of the current statement the next statement in the program is to be considered.
- **Selection**: conditional branching of the execution.

```
IF condition THEN statement ENDIF
```

In the structure `condition` is the Boolean expression (true/false).

```
IF a<b THEN a:=b+1 ENDIF

IF a<min THEN min:=a ENDIF

IF days=28 OR days=29 THEN month=February ENDIF
```

Parts of the algorithm: control statements (cont.)

Control statements:

- **Selection**: suitable operators are needed to compute the Boolean value of the expression: true (T) or false (F).

Logical operators:

AND OR NOT

`a AND b` `a OR b` `NOT a`

a	b	a AND b	a OR b	NOT a
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

Comparative operators:

equal =	greater >	smaller <
inequal <>	greater or equal >=	smaller or equal <=

`a=b` `a>b`

Parts of the algorithm: control statements (cont.)

Control statements:

- **Selection**: conditional branching of the execution.

```
IF condition THEN statement1 ELSE statement2 ENDIF
```

For example (usually intended for more comfortable reading)

```
IF x<y THEN
    min:=x
ELSE
    min:=y
ENDIF
```

Parts of the algorithm: control statements (cont.)

- Selection: the CASE structure instead of the IF structure with multiple branches.

```
CASE variable OF
   case 1: statement 1
   case 2: statement 2
   ...
   case n: statement n
   OTHER:  statement n+1
ENDCASE
```

Simpler than:

```
IF case 1 THEN statement 1 ELSE IF case 2
THEN statement 2 ELSE IF … ENDIF
```

Parts of the algorithm: control statements (cont.)

Example: the CASE statement to define the number of days of a given month

```
CASE month OF
   April, June, September, November: days:=30
   February:
      IF the year is a leap year THEN
         days:=29
      ELSE
         days:=28
      ENDIF
   OTHER: days:=31
ENDCASE
```

Parts of the algorithm: control statements (cont.)

- **Repetition** (loop): repeating certain sentences
  - a desired number of times (definite repetition) or
  - an unknown number of times (indefinite repetition).
- Definite repetition (simple and stepping):

```
REPEAT N TIMES
   statement (* several statements also possible *)
ENDREPEAT

FOR each item in the list L DO
   statement
ENDFOR

FOR i:=1,2,...,N DO
   statement
ENDFOR
```

Parts of the algorithm: control statements (cont.)

Example: What does the following FOR structure do?

```
k:=1
FOR i:=1,2,...,n DO
    k:=k*i
ENDFOR
```

In case of `n=3:`

k i "new k"	
1	in the beginning
1 1 1*1=1	the 1st round of the loop
1 2 1*2=2	the 2nd round of the loop
2 3 2*3=6	the 3rd round of the loop
6 3	at the end

- The algorithm computes the factorial of `n` as the value of the variable `k`.
- The factorial is defined as follows: `n!=n*(n-1)*(n-2)…*2*1`
- `3! = 3*2*1 = 6`

Parts of the algorithm: control statements (cont.)

- Indefinite repetition (beforehand not known how many times of repetitions).

  Pre-tested loop:

```
WHILE condition DO
    statement
ENDWHILE
```

  Post-test loop:

```
REPEAT
    statement
UNTIL condition
```

Parts of the algorithm: control statements (cont.)

Example: do the following algorithms work in the same way?

```
k:=1
i:=1
WHILE i<=n DO
    k:=k*i
    i:=i+1
ENDWHILE
```

```
k:=1
i:=1
REPEAT
    k:=k*i
    i:=i+1
UNTIL i>n
```

- The both seem to compute the factorial of `n` as `k`.
- Is there any difference?
- In the REPEAT structure, the statements in the loop are always executed at least once.
- Thus, also in case of `n` is `0`. Fortunately, 0!=1.

Example: finding the address

Given the name of a person, find the corresponding address from the list which consists of the names and the addresses.

```
WHILE the given name has not been found AND the list is
not at the end DO
   Select the next person from the list
   IF the selected name = the name to be found THEN
      Pick up the corresponding address
   ENDIF
ENDWHILE
```

- Although the number of the names in the list was known, the number of repetitions needed would not be known since it is not known where the name is in the list (if there at all).
- Note that the condition in the WHILE statement can contain many logical expressions.

Imperative paradigm: modularity

Modularity:

- **Program** ≈ a programmed algorithm: a set of the modules where one of them is the *main module* which starts the program.

```
MODULE main()
   body
ENDMODULE
```

```
MODULE main()
   x := factorial(5)
   print(x)
ENDMODULE
```

- **Module:** an “independent” and preferably general-purpose entity, consisting of statements that other modules can call.
- In the example the main module `main` calls the module `factorial` which computes the factorial of a given number.
- The factorial is defined as `n!=n*(n-1)*(n-2)…*2*1` where `n` is a given input (here 5).
- The other module in the example is `print`.

Imperative paradigm: modularity (cont.)

- **Parameter(s):** the input to a module and the output from a module.
- **Procedure**: a module which does not return any results (as output parameters) to the calling module.
  - For example: in the module "Start washing" the input could be the amount of washing liquid, but the module would not return any results as output parameters.
- **Function**: a module which returns results (as output parameters) to the calling module.
  - In our previous example, `factorial(x)` where calling the function `factorial(5)` returns 120 as the output value.
  - `x` is *a formal parameter* (defines the used variable).
  - `5` is *an actual parameter* (gives the value according to the type of the variable, here an integer).

Example: does the following pseudocode algorithm function correctly in all cases?

The goal of the module `product` is to multiply two non-negative integers x and y so that x times of the value y is summed to the product.

```
MODULE product(x,y)
    t:=y
    l:=1
    WHILE l<x DO
        t:=t+y
        l:=l+1
    ENDWHILE
    RETURN t
ENDMODULE
```

- What are the input parameters?
    - `x` and `y`.
- Is this a procedure or a function?
    - The statement `RETURN t` outputs the product as `t`.

  => the module is a function.
- Does the algorithm function in all cases?
    - Justify your reply!

Example: does not work always right!

Why?:

The algorithm cannot compute the product correctly when `x=0` since the variable `t` is set (`t:=y`) in the beginning independently of the value of `x`. Thus, instead of giving 0, the algorithm gives `y` since the loop is skipped.

The conditional statement is needed:

```
IF condition THEN statement ENDIF
IF condition THEN statement1 ELSE statement2 ENDIF
```

What kind of the conditional statement is needed? Where?

```
IF X=0 THEN RETURN 0 ELSE … ENDIF
```

Thinking futher: how about negative values as the input?

Summary

- **An algorithm** is an unambiguous set of statements (instructions) for solving a problem.
- **The imperative paradigm** refers to a representation of an algorithm that is based on a natural language or the use of a (formal) language in which statements are imperatives.
- **A pseudo-language** is a “semi-formal” language for studying the structures of algorithms. It is not directly suited for practical programming, but it is effective to practice the principles of algorithms and programming.